Especificação da linguagem Lang

Construção de Compiladores I

Prof. Rodrigo Ribeiro

28-10-2023

1 - Introdução

Neste documento é apresentada a especificação da linguagem de programação que será usada na implementação dos trabalhos da disciplina BCC328 - Construção de Compiladores I, denominada *lang*. A linguagem *lang* tem propósito meramente educacional, contendo construções que se assemelham a de várias linguagens conhecidas. No entanto, *lang* não é um subconjunto de nenhuma delas. A descrição da linguagem é dividida entre os diversos aspectos da linguagem, a saber, léxico, sintático, sistema de tipos e semântico.

É importante ressaltar que este documento serve como **referência** da linguagem e não constitui o enunciado dos trabalhos a serem desenvolvidos. O enunciado dos trabalhos envolverá subtarefas da implementação desta linguagem.

2 - Estrutura sintática

A Gramática 1 apresenta a gramática livre-de-contexto que descreve a sintaxe da linguagem *lang*. Os meta-símbolos estão entre aspas simples, quando usados como tokens. As palavras reservadas estão em negrito e os demais tokens escritos com letras maiúsculas. A notação `[ E ]` denota que `E` é **opcional**, isto é, ou a gramática derivará o conteúdo entre chaves ou λ. Por sua vez, a notação `{ E }` representa 0 ou mais ocorrências de `E`.

Em linhas gerais, um programa nesta linguagem é constituído por um conjunto de definições que podem ser tipos de dados ou funções. A estrutura sintática da linguagem é dividida em: *tipos de dados e declarações*, *funções*, *comandos* e *expressões*. Cada uma dessas estruturas são detalhadas nas subseções subsequentes.

A seguir descrevemos alguns elementos da sintaxe de *lang* utilizando exemplos.

2.1 - Tipos de dados e declarações

Programas *lang* podem conter definições de tipos de dados registro, os quais são definidos usando a palavra-chave **data**. Após a palavra-chave **data**, segue o nome do novo tipo, o qual deve começar com uma letra maiúscula, e uma lista de declarações de variáveis delimitadas por chaves. Por exemplo, um tipo para representar um número racional pode ser definido como apresentado na Figura 1:

prog	→	{def}
def	→	data | fun
data	→	**data** ID ‘{’ {decl} ‘}’
decl	→	ID ‘::’ type ‘;’
func	→	ID ‘(’ [params] ‘)’ [‘:’ type (‘,’ type)*] ‘{’ {cmd} ‘}’
params	→	ID ‘::’ type {‘,’ ID ‘::’ type}
type	→	type ‘[’ ‘]’
	|	btype
btype	→	**Int**
	|	**Char**
	|	**Bool**
	|	**Float**
	|	ID
cmd	→	‘{’ {cmd} ‘}’
	|	**if** ‘(’ exp ‘)’ cmd
	|	**if** ‘(’ exp ‘)’ cmd **else** cmd
	|	**iterate** ‘(’ exp ‘)’ cmd
	|	**read** lvalue ‘;’
	|	**print** exp ‘;’
	|	**return** exp {‘,’ exp} ‘;’
	|	lvalue = exp ‘;’
	|	ID = exp ‘;’
	|	ID ‘(’ [exps] ‘)’ [‘<’ lvalue {‘,’ lvalue} ‘>’] ‘;’
exp	→	exp && exp
	|	exp < exp
	|	exp == exp
	|	exp **!**= exp
	|	exp + exp
	|	exp **-** exp
	|	exp ***** exp
	|	exp / exp
	|	exp **%** exp
	|	! exp
	|	- exp
	|	**true**
	|	**false**
	|	**null**
	|	INT
	|	FLOAT
	|	CHAR
	|	lvalue
	|	‘(’ exp ‘)’
	|	**new** type { ‘[’ exp ‘]’ }
	|	ID ‘(’ [exps] ‘)’ ‘[’ exp ‘]’
lvalue	→	ID
	|	lvalue ‘[’ exp ‘]’
	|	lvalue **.** ID
exps	→	exp { **,** exp }

Gramática 1: Sintaxe da linguagem *lang*

```
data Racional {
   numerador :: Int;
   denominador :: Int;
}
```

Figure 1: Exemplo de tipos de dados em Lang.

Esse tipo é denominado *Racional* e contém dois atributos do tipo inteiro, um nomeado de *numerador* e o outro de *denominador*. A sintaxe para especificar os atributos de um tipo registro é a mesma usada para declarações de variáveis de funções, i.e., nome do atributo ou variável seguido por dois pontos, o tipo do atributo (variável) e finalizado com um ponto e vírgula.

2.2 - Funções

A definição de funções e procedimentos é feita dando-se o nome da função (ou procedimento) e a sua lista de parâmetros, delimitados por parêntesis. Parâmetros são seguidos por dois pontos e os tipos de retorno, em caso de função. Note que uma função pode ter mais de um retorno, os quais são separados por vírgula. Para procedimentos, não há informação sobre retorno, visto que procedimentos não retornam valores. Por fim, segue o bloco de comandos. Um exemplo de programa na linguagem *lang* é apresentado a seguir, o qual contém a definição de um procedimento denominado *main* e as funções *fat* e *divmod*. A função *fat* recebe um valor inteiro como parâmetro e tem como retorno o fatorial desse valor. A função *divmod* é um exemplo de função com mais de um valor de retorno. Esta função recebe dois parâmetros inteiros e retorna o quociente e o resto da divisão do primeiro pelo segundo parâmetro.

```
main() {
  print fat(10)[0];
}

fat(num :: Int) : Int {
  if (num < 1)
    return 1;
  else
    return num * fat(num-1)[0];
}

divmod(num :: Int, div :: Int) : Int, Int {
  q = num / div;
  r = num % div;
  return q, r;
}
```

2.3 - Comandos

A linguagem *lang* apresenta apenas 8 comandos básico, classificados em comandos de atribuição, seleção, entrada e saída, retorno, iteração e chamada de funções e procedimentos.

O comando de atribuição tem a mesma sintaxe das linguagens imperativas C/C++ e Java, na qual uma expressão do lado esquerdo especifica o endereço que será armazenado o valor resultante da avaliação da expressão do lado direito. A linguagem apresenta dois comados de seleção: um *if-then*

e *if-then-else*. Leitura e escrita da entrada/saída padrão são realizadas usando os comandos **read** e **print**, respectivamente. O comando *read* é seguido por um endereço no qual será armazenado o valor lido da entrada padrão e o comando *print* é seguido por uma expressão. Os valores de retorno de uma função são definidos por meio do comando *return*, o qual é seguido por uma lista de expressões, separadas por vírgula. A linguagem *lang* apresenta apenas um comando de iteração com a seguinte estrutura:

```
iterate (expr) cmd
```

O comando **iterate** especifica um trecho de código que será executado por uma quantidade de vezes determinada pela avaliação da expressão delimitada entre parêntesis. Ressalta-se que a expressão é avaliada uma única vez e o laço só será executado se o valor resultante da avaliação da expressão for maior que zero.

Chamadas de funções e procedimentos são comandos. A sintaxe para chamada de procedimento é o nome do procedimento seguido por uma lista de expressões separadas por vírgulas. Por exemplo, a chamada ao procedimento *main* seria:

```
main();
```

A chamada de função é similar, no entanto deve-se especificar uma lista de endereços para armazenar os valores de retorno da função, como a seguinte chamada à função *divmod*:

```
divmod(5,2)<q, r>;
```

Este comando define que os valores de retorno da função serão atribuídos as variáveis q e r.

Por fim, um bloco de comandos é definido delimitando-se uma sequência de zero ou mais comandos por chaves.

2.4 - Expressões

Expressões são abstrações sobre valores e, em *lang*, são muito semelhantes as expressões aritméticas de outras linguagens, i.e., possuem os operadores aritméticos usuais (+,-,*,/,%) além de operadores lógicos (&&, !), de comparação (<,==,!=) e valores (inteiros, caracteres, booleanos, floats, registros, vetores e chamadas de funções). Adicionalmente, parêntesis podem ser usados para determinar a **prioridade** de uma sub-expressão.

Observe, no entanto, que o conjunto de operadores é reduzido. Por exemplo, operações com valores lógicos (tipo booleano) são realizadas com os operadores de conjunção (&&) e negação (!). A linguagem não provê operadores para as demais operações lógicas. Como consequência, se queremos realizar uma seleção quando o valor de ao menos uma de duas expressões, p e q, resulta em verdadeira, escrevemos:

```
if(!(!p && !q)) { ... }
```

As chamadas de funções são expressões. Porém, diferentemente das linguagens convencionais, usa-se um índice para determinar qual dos valores de retorno da função será usado. Assim, a expressão $divmod(5, 2)[0]$ se refere ao primeiro retorno, enquanto a expressão $divmod(5, 2)[1]$ ao segundo. Note que a indexação dos retornos da função é feita de maneira análoga ao acesso de vetores, no qual o primeiro retorno é indexado por 0, o segundo por 1 e assim sucessivamente.

Nível	Operador	Descrição	Associatividade
7	[ ] . ( )	acesso a vetores acesso aos registros parêntesis	esquerda
6	! -	negação lógica menos unário	direita
5	* / %	multiplicação divisão resto	esquerda
4	+ -	adição subtração	esquerda
3	<	relacional	não é associativo
2	== !=	igualdade diferença	esquerda
1	&&	conjunção	esquerda

Tabela 1: Tabela de associatividade e precedência dos operadores. Tem a maior precedência o operador de maior nível.

A expressão $x * x + 1 < fat(2 * x)[0]$ contém operadores lógicos aritméticos e chamadas de funções. Porém, em qual ordem as operações devem ser realizadas? Se seguirmos a convenção adotada pela aritmética, primeiramente deve ser resolvidas as **operações mais fortes** ou de **maior precedência**, i.e. a multiplicação e a divisão, seguida das **operações mais fracas** ou de **menor precedência**, i.e. a soma e subtração. Assim certos operadores tem prioridade em relação a outros operadores, i.e., devem ser resolvidos antes de outros. Para a expressão $x * x + 1$ é fácil ver que a expressão $x * x$ deve ser resolvida primeiro e em seguida deve-se somar 1 ao resultado. E quando há operadores de tipos diferentes, como na expressão $x*x+1 < fat(2*x)[0]$? A situação é semelhante, resolve-se o que tem maior precedência, a qual é determinada pela linguagem. Neste exemplo, a última operação a ser realizada é a operação de comparação, cuja precedência é a menor dentre todos os operadores da expressão. A Tabela 1 apresenta a precedência dos operadores da linguagem *lang*. O operador que tiver o maior valor da coluna *nível* tem maior precedência.

Sabendo a precedência dos operadores podemos determinar em qual ordem as operações devem ser executadas quando há operadores com diferentes níveis de precedência. Entretanto, como determinar a ordem das operações se uma determinada expressão contém diferentes operadores com a mesma precedência, como nas expressões $v[3].y[0]$ e $x/3 * y$?

Em situações como essas, determinamos a ordem de avaliação das operações a partir da associatividade dos operadores, que pode ser à esquerda ou à direita. Quando os operadores são associativos à esquerda, resolvemos a ordem das operações da esquerda para a direita. Caso os operadores sejam associativos à direita, fazemos o inverso. Em ambas as expressões $v[3].y[0]$ e $x/3 * y$, os operadores são associativos à esquerda. Portanto, na primeira expressão, primeiro é realizado o acesso ao vetor v, depois acesso ao membro y e, por fim, acesso ao vetor de y. Na segunda expressão, realiza-se primeiro a divisão de x por 3 e, em seguida, a multiplicação do resultado por y.

2.5 - Estrutura léxica

A linguagem usa o conjunto de caracteres da tabela ASCII. Cada uma das possíveis categorias léxicas da linguagem são descritas a seguir:

- Um **identificador** é uma sequência de letras, dígitos e sobrescritos (*underscore*) que, obrigatoriamente, começa com uma letra minúsculas. Exemplos de identificadores: *var*, *var1* e *fun10*;
- Um **nome de tipo** é semelhante a regra de identificadores, porém a primeira letra é maiúscula; Exemplos de nomes de tipos: *Racional* e *Point*;
- Um **literal inteiro** é uma sequência de um ou mais dígitos;
- Um **literal ponto flutuante** é uma sequência de zero ou mais digitos, seguido por um ponto e uma sequência de um ou mais digitos. Exemplos de literais ponto flutuante: 3.141526535, 1.0 e .12345;
- Um **literal caractere** é um único caractere delimitado por aspas simples. Os caracteres especiais quebra-de-linha, tabulação, *backspace* e *carriage return* são definidos usando os caracteres de escape \n, \t, \b e \r, respectivamente. Para especificar um caractere \, é usado \\e para a aspas simples o \'. Exemplos de literais caractere: 'a', '\n', '\t' e '\\';
- Um **literal lógico** é um dos valores **true** ou **false**;
- O **literal nulo** é **null**;
- Os símbolos usados para **operadores** e **separadores** são (,), [,], {, }, >, ;, :, ::, ., „ =, <, ==, **!**=, +, -, *, /, %, && e !.

Todos os nomes de tipos, comandos e literais são palavras reservadas pela linguagem. O comentário de uma linha começa com `--` e se estende até a quebra de linha. A linguagem não suporta comentários com múltiplas linhas.